deer Documentation

Release 0.2

deer contributors

May 10, 2016

Contents

DeeR (Deep Reinforcement) is a python library to train an agent how to behave in a given environement so as to maximize a cumulative sum of rewards. It is based on the original deep Q learning algorithm described in : Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533. (see *What is deep reinforcement learning?*)

Here are key advantages of the library:

- Contrary to the original code, this package provides a more general framework where observations are made up of any number of elements : scalars, vectors and frames (instead of one type of frame only in the above mentionned paper). The belief state on which the agent is based to build the Q function is made up of any length history of each element provided in the observation.
- You can easily add up a validation phase that allows to stop the training process before overfitting. This possibility is useful when the environment is dependent on scarce data (e.g. limited time series).
- You also have access to advanced techniques such as Double Q-learning and prioritized Experience Replay that are readily available in the library.

In addition, the framework is made in such a way that it is easy to

- build any environment
- modify any part of the learning process
- use your favorite python-based framework to code your own neural network architecture. The provided neural network architectures are based on Theano but you may easily use another one.

It is a work in progress and input is welcome. Please submit any contribution via pull request.

CHAPTER 1

What is new

1.1 Version 0.2

- Standalone python package (you can simply do `pip install deer`)
- Integration of new examples environments : *The pendulum on a cart*, *PLE environment* and *Gym environment*
- Double Q-learning and prioritized Experience Replay
- Augmented documentation
- First automated tests

1.2 Future extensions:

- Add planning (e.g. MCTS based when deterministic environment)
- Several agents interacting in the same environment
- ...

CHAPTER 2

User Guide

2.1 Installation

2.1.1 Dependencies

This framework is tested to work under Python 2.7, and Python 3.5. It should also work with Python 3.3 and 3.4.

The required dependencies are NumPy >= 1.10, joblib >= 0.9. You also need theano >= 0.7 (lasagne is optional) or you can write your own neural network using your favorite framework.

For running some of the examples, Matplotlib >= 1.1.1 is required. You also sometimes need to install specific dependencies (e.g. for the atari games, you need to install ALE >= 0.4).

2.1.2 User install instructions

The easiest is to install the framework with pip:

```
pip install deer
```

2.1.3 Developer install instructions

As a developer, you can set you up with the bleeding-edge version of DeeR with:

```
git clone -b master https://github.com/VinF/deer.git
```

Assuming you already have a python environment with `pip`, you can automatically install all the dependencies (except specific dependencies that you may need for some examples) with:

```
pip install -r requirements.txt
```

And you can install the framework as a package using the mode `develop` so that you can make modifications and test without having to re-install the package.

```
python setup.py develop
```

2.2 Tutorial

2.2.1 What is deep reinforcement learning?

Deep reinforcement learning is the combination of two fields:

- *Reinforcement learning (RL)* is a theory that allows an agent to learn a startegy so as to maximize a sum of cumulated (delayed) rewards from any given environement. If you are not familiar with RL, you can get up to speed easily with by Sutton and Barto.
- *Deep learning* is a branch of machine learning for regression and classification. It is particularly well suited to model high-level abstractions in data by using multiple processing layers composed of multiple non-linear transformations.

This combination allows to learn complex tasks such as playing ATARI games from high-dimensional sensory inputs. For more informations, you can refer to one of the main paper in the domain : .

2.2.2 How can I get started?

First, make sure you have installed the package properly by following the steps descibed in *Installation*.

The general idea of this framework is that you need to instantiate an agent (along with a q-network) and an environment. In order to perform an experiment, you also need to attach to the agent some controllers for controlling the training and the various parameters of your agent.

The environment should be built specifically for any specific task while q-networks, the DQN agent and many controllers are provided within this package.

The best to get started is to have a look at the *Examples* and in particular the two first environments that are simple to understand:

- *Toy environment with time series*
- *The pendulum on a cart*

If you find something that is not yet implemented and if you wish to contribute, you can check the section *Development*.

2.3 Examples

You can find these examples at the . For each example at least two files are provided:

- A launcher file (whose name usually starts by `run_`).
- An environnement file (whose name usually ends by `_env`).

The launcher file performs different actions:

- It instantiates the environment, the agent (along with a q-network).
- It binds controllers to the agent
- it finally runs the experiment

Examples are better than precepts and the best is to get started with the following examples (with the simplest examples listed first)

2.3.1 `Toy environment with time series`

Description

The environment simulates the possibility of buying or selling a good. The agent can either have one unit or zero unit of that good. At each transaction with the market, the agent obtains a reward equivalent to the price of the good when selling it and the opposite when buying. In addition, a penalty of 0.5 (negative reward) is added for each transaction.

This example is defined in `examples/toy_env`. The state of the agent is made up of a past history of two punctual observations:

- The price signal
- Either the agent possesses the good or not (1 or 0)

It is important to note that the agent has no direct information about the future price signal.

Two actions are possible for the agent:

- Action 0 corresponds to selling if the agent possesses one unit or idle if the agent possesses zero unit.
- Action 1 corresponds to buying if the agent possesses zero unit or idle if the agent already possesses one unit.

The price pattern is made by repeating the following signal plus a random constant between 0 and 3:

The price signal is built following the same rules for the training and the validation environments which allows the agent to learn a strategy that exploits this successfully.

Results

Navigate to the folder `examples/toy_env` in a terminal window. The example can then be run by using

```
python run_toy_env.py
```

After 10 epochs, the following graph is obtained:

In this graph, you can see that the agent has already successfully learned to take advantage of the price pattern to buy when it is low and to sell when it is high. This example is of course easy due to the fact that the patterns are very systematic which allows the agent to successfuly learn it. It is important to note that the results shown are made on a validation set that is different from the training and we can see that learning generalizes well. For instance, the action of buying at time step 7 and 16 is the expected result because in average this will allow to make profit since the agent has no information on the future.

2.3.2 `The pendulum on a cart`

Description

The environment simulates the behavior of an inverted pendulum. The theoretical system with its equations are as described in :

- A cart of mass M that can move horizontally;
- A pole of mass m and length l attached to the cart, with θ in $[0, -\pi]$ for the lefthand plane, and $[0, \pi]$ for the righthand side. We are supposing that the cart is moving on a rail and the pole can go under it.

The goal of the agent is to balance the pole above its supporting cart ($\theta = 0$), by displacing the cart left or right - thus, 2 actions are possible. To do so, the environment communicates to the agent:

- A vector (position, speed, angle, angular speed);

- The reward associated to the action chosen by the agent.

Results

In a terminal windown go to the folder `examples/pendulum`. The example can then be run with

```
python run_pendulum.py
```

Here are the outputs of the agent after respectively 20 and 70 learning epochs, with 1000 steps in each. We clearly see the final success of the agent in controlling the inverted pendulum.

Note: a MP4 is generated every *PERIOD_BTW_SUMMARY_PERFS* epochs and you need the [FFmpeg](https://www.ffmpeg.org/) library to do so. If you do not want to install this library or to generate the videos, just set *PERIOD_BTW_SUMMARY_PERFS = -1*.

Details on the implementation

The main focus in the environment is to implement *act(self, action)* which specifies how the cart-pole system behaves in response to an input action. So first, we transcript the physical laws that rule the motion of the pole and the cart. The simulation timestep of the agent is $\Delta_t = 0.02$ second. But we discretize this value even further in *act(self, action)*, in order to obtain dynamics that are closer to the exact differential equations. Secondly, we chose the reward function as the sum of :

- $-|\theta|$ such that the agent receives 0 when the pole is standing up, and a negative reward proportional to the angle otherwise.
- $-\frac{|x|}{2}$ such that the agent receives a negative reward when it is far from $x = 0$.

2.3.3 `Gym environment`

Some examples are also provided with the .

Here is the resulting policy for the mountain car example:

Here is the resulting policy for the pendulum example:

2.3.4 `Two storage devices environment`

This example simulates the operation of a realistic micro-grid (such as a smart home for instance) that is not connected to the main utility grid (off-grid) and that is provided with PV panels, batteries and hydrogen storage. The battery has the advantage that it is not limited in instaneous power that it can provide or store. The hydrogen storage has the advantage that is can store very large quantity of energy.

```
python run_MG_two_storage_devices
```

This example uses the environment defined in MG_two_storage_devices_env.py. The agent can either choose to store in the long term storage or take energy out of it while the short term storage handle at best the lack or surplus of energy by discharging itself or charging itself respectively. Whenever the short term storage is empty and cannot handle the net demand a penalty (negative reward) is obtained equal to the value of loss load set to 2euro/kWh.

The state of the agent is made up of an history of two to four punctual observations:

- Charging state of the short term storage (0 is empty, 1 is full)
- Production and consumption (0 is no production or consumption, 1 is maximal production or consumption)
- (Distance to equinox)

- (Predictions of future production : average of the production for the next 24 hours and 48 hours)

Two actions are possible for the agent:

- Action 0 corresponds to discharging the long-term storage
- Action 1 corresponds to charging the long-term storage

More information can be found in the paper to be published : Efficient decision making in stochastic micro-grids using deep reinforcement learning, Vincent François-Lavet, David Taralla, Damien Ernst, Raphael Fonteneau

2.3.5 `ALE environment`

This environment is an interface with the that simulates any ATARI game. The hyper-parameters used in the example provided aim to simulate as closely as possible the following paper : Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

Some changes are still necessary to obtain the same performances.

2.3.6 `PLE environment`

This environment is an interface with the .

2.4 Development

DeeR is a work in progress and contributions are welcome via pull request.

For more information, you can check out this link : .

You should also make sure that you install the repository approriately for development (see *Developer install instructions*).

CHAPTER 3

API reference

If you are looking for information on a specific function, class or method, this API is for you.

3.1 `Agents`

This module contains classes used to define any agent wrapping a DQN.

NeuralAgent(environment, q_network, ...[, ...])	The NeuralAgent class wraps a deep Q-network for training and testing in a give
DataSet(env[, randomState, maxSize, ...])	A replay memory consisting of circular buffers for observations, actions, reward

3.1.1 Detailed description

class `deer.agent.`**`NeuralAgent`**(*environment*, *q_network*, *replay_memory_size*, *replay_start_size*, *batch_size*, *randomState*, *exp_priority=0*)

The NeuralAgent class wraps a deep Q-network for training and testing in a given environment.

Attach controllers to it in order to conduct an experiment (when to train the agent, when to test,...).

Parameters **environment** : object from class Environment

The environment in which the agent interacts

q_network : object from class QNetwork

The q_network associated to the agent

replay_memory_size : int

Size of the replay memory

replay_start_size : int

Number of observations (=number of time steps taken) in the replay memory before starting learning

batch_size : int

Number of tuples taken into account for each iteration of gradient descent

randomState : numpy random number generator

Seed

exp_priority : float, optional

The exponent that determines how much prioritization is used, default is 0 (uniform priority). One may check out Schaul et al. (2016) - Prioritized Experience Replay.

Methods

`attach`(controller)	
`avgBellmanResidual`()	Returns the average training loss on the epoch
`avgEpisodeVValue`()	Returns the average V value on the episode
`bestAction`()	Returns the best Action
`detach`(controllerIdx)	
`discountFactor`()	Get the discount factor
`dumpNetwork`(fname, nEpoch)	
`epsilon`()	Get the epsilon for ϵ-greedy exploration
`learningRate`()	Get the learning rate
`mode`()	
`overrideNextAction`(action)	Possibility to override the chosen action.
`resumeTrainingMode`()	
`run`(nEpochs, epochLength)	
`setControllersActive`(toDisable, active)	Activate controller
`setDiscountFactor`(df)	Set the discount factor
`setEpsilon`(e)	Set the epsilon used for ϵ-greedy exploration
`setLearningRate`(lr)	Set the learning rate for the gradient descent
`startMode`(mode, epochLength)	
`summarizeTestPerformance`()	
`totalRewardOverLastTest`()	Returns the average sum of reward per episode
`train`()	

avgBellmanResidual()
Returns the average training loss on the epoch

avgEpisodeVValue()
Returns the average V value on the episode

bestAction()
Returns the best Action

discountFactor()
Get the discount factor

epsilon()
Get the epsilon for ϵ-greedy exploration

learningRate()
Get the learning rate

overrideNextAction(*action*)
Possibility to override the chosen action. This possibility should be used on the signal OnActionChosen.

setControllersActive(*toDisable*, *active*)
Activate controller

setDiscountFactor(*df*)
Set the discount factor

setEpsilon(*e*)
Set the epsilon used for ϵ-greedy exploration

setLearningRate(*lr*)
Set the learning rate for the gradient descent

totalRewardOverLastTest()
Returns the average sum of reward per episode

class deer.agent.**DataSet**(*env*, *randomState=None*, *maxSize=1000*, *use_priority=False*)
A replay memory consisting of circular buffers for observations, actions, rewards and terminals.

Methods

actions()	Get all actions currently in the replay memory, ordered by time where they we
addSample(obs, action, reward, isTerminal, ...)	Store a (observation[for all subjects], action, reward, isTerminal) in the datase
nElems()	Get the number of samples in this dataset (i.e.
observations()	Get all observations currently in the replay memory, ordered by time where th
randomBatch(size, use_priority)	Return corresponding states, actions, rewards, terminal status, and next_states
rewards()	Get all rewards currently in the replay memory, ordered by time where they w
terminals()	Get all terminals currently in the replay memory, ordered by time where they
update_priorities(priorities, rndValidIndices)	

actions()
Get all actions currently in the replay memory, ordered by time where they were taken.

addSample(*obs*, *action*, *reward*, *isTerminal*, *priority*)
Store a (observation[for all subjects], action, reward, isTerminal) in the dataset.

Parameters **obs** : ndarray

An ndarray(dtype='object') where obs[s] corresponds to the observation made on subject s before the agent took action [action].

action : int

The action taken after having observed [obs].

reward : float

The reward associated to taking this [action].

isTerminal : bool

Tells whether [action] lead to a terminal state (i.e. corresponded to a terminal transition).

priority : float

The priority to be associated with the sample

nElems()
Get the number of samples in this dataset (i.e. the current memory replay size).

observations()
Get all observations currently in the replay memory, ordered by time where they were observed.

observations[s][i] corresponds to the observation made on subject s before the agent took actions()[i].

randomBatch(*size*, *use_priority*)
Return corresponding states, actions, rewards, terminal status, and next_states for size randomly chosen transitions. Note that if terminal[i] == True, then next_states[s][i] == np.zeros_like(states[s][i]) for each subject s.

Parameters **size** : int

Number of transitions to return.

Returns **states** : ndarray

An ndarray(size=number_of_subjects, dtype='object), where states[s] is a 2+D matrix of dimensions size x s.memorySize x "shape of a given observation for this subject". States were taken randomly in the data with the only constraint that they are complete regarding the histories for each observed subject.

actions : ndarray

An ndarray(size=number_of_subjects, dtype='int32') where actions[i] is the action taken after having observed states[:][i].

rewards : ndarray

An ndarray(size=number_of_subjects, dtype='float32') where rewards[i] is the reward obtained for taking actions[i-1].

next_states : ndarray

Same structure than states, but next_states[s][i] is guaranteed to be the information concerning the state following the one described by states[s][i] for each subject s.

terminals : ndarray

An ndarray(size=number_of_subjects, dtype='bool') where terminals[i] is True if actions[i] lead to terminal states and False otherwise

`rewards`()

Get all rewards currently in the replay memory, ordered by time where they were received.

`terminals`()

Get all terminals currently in the replay memory, ordered by time where they were observed.

terminals[i] is True if actions()[i] lead to a terminal state (i.e. corresponded to a terminal transition), and False otherwise.

`update_priorities`(*priorities*, *rndValidIndices*)

3.2 Controllers

This file defines the base Controller class and some presets controllers that you can use for controlling the training and the various parameters of your agents.

Controllers can be attached to an agent using the agent's `attach(Controller)` method. The order in which controllers are attached matters. Indeed, if controllers C1, C2 and C3 were attached in this order and C1 and C3 both listen to the OnEpisodeEnd signal, the OnEpisodeEnd() method of C1 will be called *before* the OnEpisodeEnd() method of C3, whenever an episode ends.

`Controller`()	A base controller that does nothing when receiving the various signals emit
`LearningRateController`([...])	A controller that modifies the learning rate periodically upon epochs end.
`EpsilonController`([initialE, eDecays, eMin, ...])	A controller that modifies the probability "epsilon" of taking a random actic
`DiscountFactorController`([...])	A controller that modifies the q-network discount periodically.
`TrainerController`([evaluateOn, periodicity, ...])	A controller that make the agent train on its current database periodically.
`InterleavedTestEpochController`([id, ...])	A controller that interleaves a test epoch between training epochs of the age
`FindBestController`([validationID, testID, ...])	A controller that finds the neural net performing at best in validation mode

3.2.1 Detailed description

class `deer.experiment.base_controllers.`**`Controller`**

A base controller that does nothing when receiving the various signals emitted by an agent. This class should be the base class of any controller you would want to define.

Methods

`OnActionChosen`(agent, action)	Called whenever the agent has chosen an action.
`OnActionTaken`(agent)	Called whenever the agent has taken an action on its environment.
`OnEnd`(agent)	Called when the agent has finished processing all its epochs, just before returnin
`OnEpisodeEnd`(agent, terminalReached, reward)	Called whenever the agent ends an episode, just after this episode ended and be
`OnEpochEnd`(agent)	Called whenever the agent ends an epoch, just after the last episode of this epoc
`OnStart`(agent)	Called when the agent is going to start working (before anything else).
`setActive`(active)	Activate or deactivate this controller.

`OnActionChosen`(*agent*, *action*)

Called whenever the agent has chosen an action.

This occurs after the agent state was updated with the new observation it made, but before it applied this action on the environment and before the total reward is updated.

`OnActionTaken`(*agent*)

Called whenever the agent has taken an action on its environment.

This occurs after the agent applied this action on the environment and before terminality is evaluated. This is called only once, even in the case where the agent skip frames by taking the same action multiple times. In other words, this occurs just before the next observation of the environment.

`OnEnd`(*agent*)

Called when the agent has finished processing all its epochs, just before returning from its run() method.

`OnEpisodeEnd`(*agent*, *terminalReached*, *reward*)

Called whenever the agent ends an episode, just after this episode ended and before any OnEpochEnd() signal could be sent.

Parameters **agent** : NeuralAgent

The agent firing the event

terminalReached : bool

Whether the episode ended because a terminal transition occured. This could be False if the episode was stopped because its step budget was exhausted.

reward : float

The reward obtained on the last transition performed in this episode.

`OnEpochEnd`(*agent*)

Called whenever the agent ends an epoch, just after the last episode of this epoch was ended and after any OnEpisodeEnd() signal was processed.

Parameters **agent** : NeuralAgent

The agent firing the event

`OnStart`(*agent*)

Called when the agent is going to start working (before anything else).

This corresponds to the moment where the agent's run() method is called.

Parameters **agent** : NeuralAgent

The agent firing the event

setActive(*active*)

Activate or deactivate this controller.

A controller should not react to any signal it receives as long as it is deactivated. For instance, if a controller maintains a counter on how many episodes it has seen, this counter should not be updated when this controller is disabled.

class deer.experiment.base_controllers.**LearningRateController**(*initialLearningRate=0.0002*, *learningRateDecay=1.0*, *periodicity=1*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that modifies the learning rate periodically upon epochs end.

Parameters **initialLearningRate** : float

The learning rate upon agent start

learningRateDecay : float

The factor by which the previous learning rate is multiplied every [periodicity] epochs.

periodicity : int

How many epochs are necessary before an update of the learning rate occurs

Methods

OnActionChosen(agent, action)	Called whenever the agent has chosen an action.
OnActionTaken(agent)	Called whenever the agent has taken an action on its environment.
OnEnd(agent)	Called when the agent has finished processing all its epochs, just before returnir
OnEpisodeEnd(agent, terminalReached, reward)	Called whenever the agent ends an episode, just after this episode ended and bef
OnEpochEnd(agent)	
OnStart(agent)	
setActive(active)	Activate or deactivate this controller.

class deer.experiment.base_controllers.**EpsilonController**(*initialE=1.0*, *eDecays=10000*, *eMin=0.1*, *evaluateOn='action'*, *periodicity=1*, *resetEvery='none'*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that modifies the probability "epsilon" of taking a random action periodically.

Parameters **initialE** : float

Start epsilon

eDecays : int

How many updates are necessary for epsilon to reach eMin

eMin : float

End epsilon

evaluateOn : str

After what type of event epsilon shoud be updated periodically. Possible values: 'action', 'episode', 'epoch'.

periodicity : int

How many [evaluateOn] are necessary before an update of epsilon occurs

resetEvery : str

After what type of event epsilon should be reset to its initial value. Possible values: 'none', 'episode', 'epoch'.

Methods

`OnActionChosen`(agent, action)	
`OnActionTaken`(agent)	Called whenever the agent has taken an action on its environment.
`OnEnd`(agent)	Called when the agent has finished processing all its epochs, just before returnin
`OnEpisodeEnd`(agent, terminalReached, reward)	
`OnEpochEnd`(agent)	
`OnStart`(agent)	
`setActive`(active)	Activate or deactivate this controller.

class `deer.experiment.base_controllers.`**`DiscountFactorController`**(*initialDiscountFactor=0.9*, *discountFactorGrowth=1.0*, *discountFactorMax=0.99*, *periodicity=1*)

Bases: *`deer.experiment.base_controllers.Controller`*

A controller that modifies the q-network discount periodically. More informations in : Francois-Lavet Vincent et al. (2015) - How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies (http://arxiv.org/abs/1512.02011).

Parameters **initialDiscountFactor** : float

Start discount

discountFactorGrowth : float

The factor by which the previous discount is multiplied every [periodicity] epochs.

discountFactorMax : float

Maximum reachable discount

periodicity : int

How many training epochs are necessary before an update of the discount occurs

Methods

`OnActionChosen`(agent, action)	Called whenever the agent has chosen an action.

Table 3.8 – continued from previous page

`OnActionTaken`(agent)	Called whenever the agent has taken an action on its environment.
`OnEnd`(agent)	Called when the agent has finished processing all its epochs, just before returnin
`OnEpisodeEnd`(agent, terminalReached, reward)	Called whenever the agent ends an episode, just after this episode ended and bef
`OnEpochEnd`(agent)	
`OnStart`(agent)	
`setActive`(active)	Activate or deactivate this controller.

class `deer.experiment.base_controllers.`**`TrainerController`**(*evaluateOn='action'*, *periodicity=1*, *showEpisodeAvgVValue=True*, *showAvgBellmanResidual=True*)

Bases: *`deer.experiment.base_controllers.Controller`*

A controller that make the agent train on its current database periodically.

Parameters **evaluateOn** : str

After what type of event the agent shoud be trained periodically. Possible values: 'action', 'episode', 'epoch'. The first training will occur after the first occurence of [evaluateOn].

periodicity : int

How many [evaluateOn] are necessary before a training occurs _showAvgBellmanResidual [bool] - Whether to show an informative message after each episode end (and after a training if [evaluateOn] is 'episode') about the average bellman residual of this episode

showEpisodeAvgVValue : bool

Whether to show an informative message after each episode end (and after a training if [evaluateOn] is 'episode') about the average V value of this episode

Methods

`OnActionChosen`(agent, action)	Called whenever the agent has chosen an action.
`OnActionTaken`(agent)	
`OnEnd`(agent)	Called when the agent has finished processing all its epochs, just before returnin
`OnEpisodeEnd`(agent, terminalReached, reward)	
`OnEpochEnd`(agent)	
`OnStart`(agent)	
`setActive`(active)	Activate or deactivate this controller.

class `deer.experiment.base_controllers.`**`InterleavedTestEpochController`**(*id=0*, *epochLength=500*, *controllersToDisable=[]*, *periodicity=2*, *showScore=True*, *summarizeEvery=10*)

Bases: *`deer.experiment.base_controllers.Controller`*

A controller that interleaves a test epoch between training epochs of the agent.

Parameters **id** : int

The identifier (>= 0) of the mode each test epoch triggered by this controller will belong to. Can be used to discriminate between datasets in your Environment subclass (this is the argument that will be given to your environment's reset() method when starting the test epoch).

epochLength : float

The total number of transitions that will occur during a test epoch. This means that this epoch could feature several episodes if a terminal transition is reached before this budget is exhausted.

controllersToDisable : list of int

A list of controllers to disable when this controller wants to start a test epoch. These same controllers will be reactivated after this controller has finished dealing with its test epoch.

periodicity : int

How many epochs are necessary before a test epoch is ran (these controller's epochs included: "1 test epoch on [periodicity] epochs"). Minimum value: 2.

showScore : bool

Whether to print an informative message on stdout at the end of each test epoch, about the total reward obtained in the course of the test epoch.

summarizeEvery : int

How many of this controller's test epochs are necessary before the attached agent's summarizeTestPerformance() method is called. Give a value <= 0 for "never". If > 0, the first call will occur just after the first test epoch.

Methods

`OnActionChosen`(agent, action)	Called whenever the agent has chosen an action.
`OnActionTaken`(agent)	Called whenever the agent has taken an action on its environment.
`OnEnd`(agent)	Called when the agent has finished processing all its epochs, just before returnir
`OnEpisodeEnd`(agent, terminalReached, reward)	Called whenever the agent ends an episode, just after this episode ended and bef

Table 3.10 – continued from previous page

OnEpochEnd(agent)	
OnStart(agent)	
setActive(active)	Activate or deactivate this controller.

class deer.experiment.base_controllers.**FindBestController**(*validationID=0*, *testID=None*, *unique_fname='nnet'*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that finds the neural net performing at best in validation mode (i.e. for mode = [validationID]) and computes the associated generalization score in test mode (i.e. for mode = [testID], and this only if [testID] is different from None). This controller should never be disabled by InterleavedTestControllers as it is meant to work in conjunction with them.

At each epoch end where this controller is active, it will look at the current mode the agent is in.

If the mode matches [validationID], it will take the total reward of the agent on this epoch and compare it to its current best score. If it is better, it will ask the agent to dump its current nnet on disk and update its current best score. In all cases, it saves the validation score obtained in a vector.

If the mode matches [testID], it saves the test (= generalization) score in another vector. Note that if [testID] is None, no test mode score are ever recorded.

At the end of the experiment (OnEnd), if active, this controller will print information about the epoch at which the best neural net was found together with its generalization score, this last information shown only if [testID] is different from None. Finally it will dump a dictionnary containing the data of the plots ({n: number of epochs elapsed, ts: test scores, vs: validation scores}). Note that if [testID] is None, the value dumped for the 'ts' key is [].

Parameters **validationID** : int

See synopsis

testID : int

See synopsis

unique_fname : str

A unique filename (basename for score and network dumps).

Methods

OnActionChosen(agent, action)	Called whenever the agent has chosen an action.
OnActionTaken(agent)	Called whenever the agent has taken an action on its environment.
OnEnd(agent)	
OnEpisodeEnd(agent, terminalReached, reward)	Called whenever the agent ends an episode, just after this episode ended and be
OnEpochEnd(agent)	
OnStart(agent)	Called when the agent is going to start working (before anything else).
setActive(active)	Activate or deactivate this controller.

3.3 Environment interface

3.3.1 Detailed description

class deer.base_classes.**Environment**

All your Environment classes should inherit this interface.

The environment defines the dynamics and the reward signal that the agent observes when interacting with it.

An agent sees at any time-step from the environment a collection of observable elements. Observing the environment at time t thus corresponds to obtaining a punctual observation for each of these elements. According to the control problem to solve, it might be useful for the agent to not only take action based on the current punctual observations but rather on a collection of the last punctual observations. In this framework, it's the environment that defines the number of each punctual observation to be considered.

Different "modes" are used in this framework to allow the environment to have different dynamics and/or reward signal. For instance, in training mode, only a part of the dynamics may be available so that it is possible to see how well the agent generalizes to a slightly different one.

Methods

`act`(action)	Apply the agent action [action] on the environment.
`inTerminalState`()	Tell whether the environment reached a terminal state after the last transition (i.e.
`inputDimensions`()	Get the shape of the input space for this environment.
`nActions`()	Get the number of different actions that can be taken on this environment.
`observationType`(subject)	Get the most inner type (np.uint8, np.float32, ...) of [subject].
`observe`()	Get a list of punctual observations on all subjects composing this environment.
`reset`(mode)	Reset the environment and put it in mode [mode].
`summarizePerformance`(test_data_set)	Additional optional hook that can be used to show a summary of the performance of th

act(*action*)

Apply the agent action [action] on the environment.

Parameters action : int

The action selected by the agent to operate on the environment. Should be an identifier included between 0 included and nActions() excluded.

inTerminalState()

Tell whether the environment reached a terminal state after the last transition (i.e. the last transition that occured was terminal).

As the majority of control tasks considered have no end (a continuous control should be operated), by default this returns always False. But in the context of a video game for instance, terminal states can occurs and these cases this method should be overriden.

Returns isTerminal : bool

inputDimensions()

Get the shape of the input space for this environment.

This returns a list whose length is the number of subjects observed on the environment. Each element of the list is a tuple whose content and size depends on the type of data observed: the first integer is always the history size (or batch size) for observing this subject and the rest describes the shape of a single observation on this subject: - () or (1,) means each observation on this subject is a single number and the history size is

1 (= no history) - (N,) means each observation on this subject is a single number and the history size is N - (N, M) means each observation on this subject is a vector of length M and the history size is N - (N, M1, M2) means each observation on this subject is a matrix with M1 rows and M2 columns and the history size is N

nActions()
Get the number of different actions that can be taken on this environment.

observationType(*subject*)
Get the most inner type (np.uint8, np.float32, ...) of [subject].

Parameters subject : int

The subject

observe()
Get a list of punctual observations on all subjects composing this environment.

This returns a list where element i is a punctual observation on subject i. You will notice that the history of observations on this subject is not returned; only the very last observation. Each element is thus either a number, vector or matrix and not a succession of numbers, vectors and matrices.

See the documentation of batchDimensions() for more information about the shape of the observations according to their mathematical representation (number, vector or matrix).

reset(*mode*)
Reset the environment and put it in mode [mode].

The [mode] can be used to discriminate for instance between an agent which is training or trying to get a validation or generalization score. The mode the environment is in should always be redefined by resetting the environment using this method, meaning that the mode should be preserved until the next call to reset().

Parameters mode : int

The mode to put the environment into. Mode "-1" is reserved and always means "training".

summarizePerformance(*test_data_set*)
Additional optional hook that can be used to show a summary of the performance of the agent on the environment in the current mode (in validation and or generalization for example).

Parameters test_data_set : agent.DataSet

The dataset maintained by the agent in the current mode, which contains observations, actions taken and rewards obtained, as well as wether each transition was terminal or not. Refer to the documentation of agent.DataSet for more information.

3.4 Q-networks

deer.base_classes.QNetwork(environment, ...)	All the Q-networks classes should inherit this interface.
deer.q_networks.q_net_theano.MyQNetwork(...)	Deep Q-learning network using Theano
deer.q_networks.q_net_lasagne.MyQNetwork	

3.4.1 Detailed description

class deer.base_classes.**QNetwork**(*environment*, *batchSize*)
All the Q-networks classes should inherit this interface.

Parameters **environment** : object from class Environment

The environment linked to the Q-network

batch_size : int

Number of tuples taken into account for each iteration of gradient descent

Methods

`chooseBestAction`(state)	Get the best action for a belief state
`discountFactor`()	Getting the discount factor
`learningRate`()	Getting the learning rate
`qValues`(state)	Get the q value for one belief state
`setDiscountFactor`(df)	Setting the discount factor
`setLearningRate`(lr)	Setting the learning rate
`train`(states, actions, rewards, nextStates, ...)	This method performs the Bellman iteration for one batch of tuples.

`chooseBestAction`(*state*)
Get the best action for a belief state

`discountFactor`()
Getting the discount factor

`learningRate`()
Getting the learning rate

`qValues`(*state*)
Get the q value for one belief state

`setDiscountFactor`(*df*)
Setting the discount factor

Parameters **df** : float

The discount factor that has to bet set

`setLearningRate`(*lr*)
Setting the learning rate

Parameters **lr** : float

The learning rate that has to bet set

`train`(*states*, *actions*, *rewards*, *nextStates*, *terminals*)
This method performs the Bellman iteration for one batch of tuples.

class `deer.q_networks.q_net_theano.`**`MyQNetwork`**(*environment*, *rho*, *rms_epsilon*, *momentum*, *clip_delta*, *freeze_interval*, *batchSize*, *network_type*, *update_rule*, *batch_accumulator*, *randomState*, *DoubleQ=False*, *TheQNet=<class 'deer.q_networks.NN_theano.NN'>*)

Bases: `deer.base_classes.QNetwork.QNetwork`

Deep Q-learning network using Theano

Parameters **environment** : object from class Environment

rho : float

rms_epsilon : float

momentum : float

clip_delta : float

freeze_interval : int

batch_size : int

Number of tuples taken into account for each iteration of gradient descent

network_type : str

update_rule: str

batch_accumulator : str

randomState : numpy random number generator

DoubleQ : bool, optional

Activate or not the DoubleQ learning, default : False. More informations in : Hado van Hasselt et al. (2015) - Deep Reinforcement Learning with Double Q-learning.

TheQNet : object, optional

default is deer.qnetworks.NN_theano

Methods

chooseBestAction
discountFactor
learningRate
qValues
setDiscountFactor
setLearningRate
toDump
train

chooseBestAction (*state*)
Get the best action for a belief state

Returns **The best action** : int

qValues (*state_val*)
Get the q value for one belief state

Returns The q value for the provided belief state

train (*states_val*, *actions_val*, *rewards_val*, *next_states_val*, *terminals_val*)
Train one batch.

1. Set shared variable in states_shared, next_states_shared, actions_shared, rewards_shared, terminals_shared
2. perform batch training

Parameters **states_val** : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

actions_val : b x 1 numpy array of integers

rewards_val : b x 1 numpy array

next_states_val : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

terminals_val : b x 1 numpy boolean array (currently ignored)

Returns average loss of the batch training

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index